Parallel and Distributed Computing

Comprehensive Lecture Notes

Ref: Tanenbaum & Steen · Hwang, Dongarra & Fox · Culler & Singh
Prerequisites: Operating Systems | Credit Hours: 3

Table of Contents

1. Synchronous & Asynchronous Computation/Communication
2. Concurrency Control
3. Fault Tolerance
4. GPU Architecture and Programming
5. Heterogeneity in Distributed Systems
6. Interconnection Topologies
7. Load Balancing
8. Memory Consistency Models
9. Memory Hierarchies
10. Message Passing Interface (MPI)
11. MIMD and SIMD Architectures
12. Multithreaded Programming
13. Parallel Algorithms & Architectures
14. Parallel I/O
15. Performance Analysis and Tuning
16. Power in Parallel Systems
17. Programming Models
18. Scalability and Performance Studies
19. Scheduling
20. Storage Systems
21. Synchronization
22. Tools Reference

1. Synchronous & Asynchronous Computation / Communication

Synchronous operations block the calling process until fully complete; asynchronous operations return immediately, allowing compute and communication to overlap.

1.1 Synchronous Communication

• The sender issues a send call and is **blocked** — it cannot execute any further instruction — until the receiver has posted a matching receive and the data transfer is complete.

• Both endpoints must rendezvous at the same point in time, creating tight temporal coupling. If the receiver is busy, the sender wastes CPU cycles sitting idle at the send call.

• Correctness is easy to reason about: after send() returns, the data is guaranteed delivered. No hidden races about whether the message arrived.

• MPI blocking variants:

– **MPI_Send** — blocks until buffer is safe to reuse (may copy to internal system buffer).

– **MPI_Ssend** — strict synchronous; does NOT return until receiver has posted MPI_Recv. True rendezvous. No internal buffering.

– **MPI_Recv** — blocks until message fully received in buffer.

• Hardware implementation uses handshake signals: sender transmits data + VALID signal; receiver processes it and sends ACK; sender proceeds only after ACK. Directly maps to TCP three-way handshake.

1.2 Asynchronous Communication

• The sender **posts** a send request and immediately continues execution. The message is placed in a software or hardware buffer; the communication subsystem transmits it in the background.

• This decouples sender and receiver timing entirely. Latency of communication is hidden behind computation — the key technique for achieving **communication–computation overlap** in HPC codes.

• The sender must NOT modify the send buffer after posting until the operation completes. The hardware NIC may still be reading it via DMA. Modifying it early produces corrupted data at the receiver — a race condition that is very hard to debug.

• MPI non-blocking functions:

– **MPI_Isend(buf, count, type, dest, tag, comm, &request;)** — returns immediately with a request handle.

– **MPI_Irecv(buf, count, type, src, tag, comm, &request;)** — pre-posts a receive buffer for incoming data.

– **MPI_Wait(&request;, &status;)** — blocks until the specific operation is complete; safe to reuse buffer after this.

– **MPI_Test(&request;, &flag;, &status;)** — non-blocking check; returns flag=1 if done, flag=0 if still in progress.

– **MPI_Waitall(count, requests[], statuses[])** — waits for all operations in an array simultaneously; more efficient than looping MPI_Wait.

• When NIC hardware completes the transfer, it performs a **DMA write** directly into the pre-posted receive buffer without CPU involvement. This zero-copy path is why InfiniBand achieves ~1 µs latency and 200 Gb/s bandwidth.

1.3 Sync vs Async — Comparison

Aspect	Synchronous	Asynchronous
Blocking	Yes — caller waits until done	No — caller continues immediately
CPU idle time	High when receiver is slow	Near zero; compute runs during transfer
Buffer safety	Safe immediately after call returns	Must call MPI_Wait before reusing buffer
Correctness reasoning	Simple — sequential semantics	Complex — must track outstanding requests
Deadlock risk	Higher — two blocking sends face to face deadlock	Lower — non-blocking avoids send/recv ordering issues
Performance	Lower — compute and comm serialized	Higher — comm latency hidden behind compute
MPI example	MPI_Send + MPI_Recv	MPI_Isend + MPI_Irecv + compute + MPI_Wait

1.4 Barriers — Types and Costs

• A barrier is a global synchronization point: **all** processes/threads must arrive before any is allowed to proceed. Used to separate computation phases and ensure visibility of writes before the next read phase.

• Cost = idle time of fast processes waiting for the slowest process (the straggler). In iterative codes with 1,000 iterations and 10% imbalance, barrier overhead compounds into a large fraction of total runtime.

• Without a barrier: a fast process could read stale data that a slow process has not yet written, causing a data race.

• **Centralized barrier** — all processes report to one counter; simple but creates contention at high process counts.

• **Tree barrier** — processes arranged in binary tree:

  – Gather phase (bottom-up): leaf nodes send 'arrived' to parents; root knows all arrived when it receives from both children.

  – Broadcast phase (top-down): root sends 'release'; propagates to all leaves.

  – Total time: 2 × log■(P) message latencies. Far better than P messages in centralized approach.

• **Dissemination barrier** — in round k, process i notifies process $(i + 2^k)$ mod P.

  – After ■log■P■ rounds, every process has directly or indirectly heard from every other process.

  – No tree structure needed; works well on irregular topologies.

• MPI: **MPI_Barrier(MPI_COMM_WORLD)**. OpenMP: **#pragma omp barrier** (explicit) or implicit at end of parallel for.

• Removing unnecessary barriers is one of the highest-impact optimizations in parallel codes. Use **nowait** clause in OpenMP where the next phase does not depend on the prior phase's writes.

2. Concurrency Control

Concurrency control comprises all mechanisms that ensure correct, consistent execution when multiple processes or threads access shared resources simultaneously. Without it, programs produce non-deterministic, data-dependent bugs that are nearly impossible to reproduce.

2.1 Race Conditions

• A race condition occurs when the correctness of output depends on the timing or interleaving of two or more concurrent threads. The same program may produce correct output most runs and fail only under a specific scheduling interleaving.

• Classic example — two threads incrementing a shared counter:

– The increment requires three instructions: LOAD value → ADD 1 → STORE result.

– If both threads LOAD the same old value before either STOREs, both write old+1 instead of old+2. One increment is lost.

– The counter should increase by 2 but increases by only 1. This is a lost update.

• A **critical section** is the code region accessing shared mutable data that must execute atomically — without interleaving from other threads — to maintain correctness.

– Too large a critical section: reduces concurrency unnecessarily; other threads wait longer.

– Too small a critical section: misses some shared accesses; race still exists.

• Detection tools:

– **ThreadSanitizer (TSan)** — instruments code at compile time; reports races at runtime when actually triggered. ~5–10× overhead.

– **Helgrind (Valgrind)** — tracks lock acquisition orders; warns about potential deadlocks and races even before they trigger.

2.2 Mutual Exclusion

• Mutual exclusion (mutex) guarantees at most one thread/process executes the critical section at any instant.

• **Dijkstra's three requirements (1965)** — any correct mutex algorithm must satisfy all three:

– **Safety (Mutual Exclusion)**: at most one process in the CS at a time.

– **Liveness (Progress)**: if no process is in the CS and one or more wait, at least one waiting process must eventually enter.

– **Bounded Waiting (Starvation-freedom)**: there is a finite limit on how many times other processes can enter the CS before a waiting process is granted entry.

• **Hardware atomic primitives** — the only correct foundation on modern hardware:

– **Test-and-Set (TAS)**: atomically read memory and set it to 1; returns old value. If old value was 0, lock acquired; if 1, already held.

– **Compare-and-Swap (CAS)**: if memory[addr] == expected, replace with new value and return true; otherwise return false. More powerful — foundation of all lock-free data structures.

– **Fetch-and-Add**: atomically add a value and return old value. Used in ticket locks.

• **Spin lock**: thread retries CAS in a tight loop when lock is held. Efficient only for very short critical sections (< few microseconds). Wastes CPU on longer waits.

• **Test-and-Test-and-Set (TTAS)** optimization:

– Instead of executing expensive atomic TAS every loop iteration (generates write bus traffic), first do a plain read.

– Only attempt atomic TAS when the plain read sees the lock is free.

– Reduces coherence bus traffic by orders of magnitude when many threads are spinning.

• Software-only algorithms (Peterson's, Bakery) satisfy all three requirements but require explicit memory fences on modern hardware because CPUs reorder loads/stores.

2.3 Semaphores

• A semaphore is an integer variable with two atomic operations: **wait(S)** and **signal(S)**. Introduced by Dijkstra (1968).

• **wait(S) / P(S) / down(S)**:

– Decrement S. If S becomes negative, the calling process is blocked and added to the semaphore's wait queue.

– The decrement-and-block is atomic — no other process can interrupt midway.

• **signal(S) / V(S) / up(S)**:

– Increment S. If S was negative (meaning processes are blocked), one blocked process is removed from the queue and made runnable.

– The thread to wake is chosen by the OS (FIFO for fairness, or arbitrary).

• **Binary semaphore** — values only 0 (locked) or 1 (unlocked):

– Before entering CS: wait() — decrements to 0; blocks if already 0.

– After leaving CS: signal() — increments back to 1; wakes one waiting thread.

– Unlike a pure mutex, can be signaled by a DIFFERENT thread than the one that called wait — enables producer-consumer signaling.

• **Counting semaphore** — initialized to N (number of available resources):

– Each consumer calls wait() to take one resource; each producer calls signal() to return one.

– When count reaches 0, the next consumer blocks. No busy-waiting.

– Solves the bounded-buffer (producer-consumer) problem elegantly.

• Common errors with semaphores:

– Forgetting a signal() → permanent deadlock; process waits forever.

– Calling signal() twice → two threads enter CS simultaneously; mutual exclusion violated.

– These error-prone patterns motivate higher-level constructs like monitors.

2.4 Monitors and Condition Variables

• A monitor is a synchronization construct that encapsulates shared data with the procedures operating on it. Only one thread can be active inside the monitor at any time — implicit mutual exclusion.

• No explicit lock/unlock calls needed around every shared access. This makes concurrent programs significantly easier to write correctly compared to semaphores.

• Foundation of: Java **synchronized** keyword, Python **threading.Lock()**, C++ **std::mutex** + **std::unique_lock**.

• **Condition variables** — allow waiting inside a monitor for a specific condition:

– **cond.wait(lock)**: atomically releases the monitor lock AND suspends the calling thread on the condition's queue. Atomic release-and-suspend prevents missed wakeup.

– **cond.signal() / cond.notify_one()**: wakes exactly one waiting thread; it reacquires the lock and continues.

– **cond.notify_all()**: wakes all waiting threads; each contends for the lock one at a time.

• **Mesa semantics** (Java, POSIX, C++11) — condition waits MUST be in a while loop, not an if:

– **while (!condition) { cond.wait(lock); }**

– After wakeup, the condition may no longer be true — another thread may have changed state between the signal and the waiting thread reacquiring the lock.

– Hoare semantics guarantee condition is true immediately after signal but are harder to implement efficiently; not used in practice.

• Readers-Writers problem using monitors:

– Multiple readers can hold the read lock simultaneously — reads don't conflict.

– A writer needs exclusive access — no readers or other writers.

– Two condition variables: one for readers waiting for no active writer; one for writers waiting for no active readers/writers.

2.5 Deadlock — Four Conditions, Detection, Prevention, Avoidance

• Deadlock: a set of processes each waiting for a resource held by another in the set. Circular wait that cannot resolve without external intervention.

• **Coffman's four necessary conditions (1971)** — ALL four must hold simultaneously for deadlock:

– **Mutual Exclusion**: at least one resource held in non-sharable mode.

– **Hold and Wait**: a process holding resources is waiting to acquire more.

– **No Preemption**: resources cannot be forcibly taken from a holding process.

– **Circular Wait**: circular chain P■→P■→…→P■→P■ where each P■ waits for a resource held by P■■■.

• **Prevention** — eliminate at least one condition by design:

– Eliminate Hold-and-Wait: process requests ALL resources at start or releases all before requesting new ones. Wastes resources; may starve.

– Eliminate Circular Wait: impose a global ordering on resource types; all processes must request resources in increasing order. Most practical prevention method.

– Allow Preemption: if a process requests a held resource, preempt the holder. Only viable for resources whose state can be saved and restored (CPU, not printers).

• **Banker's Algorithm (Dijkstra)** — deadlock avoidance:

– Maintains: Max (maximum demand per process), Allocation (current allocation), Available (free resources).

– Before granting a request, checks if resulting state is 'safe': whether a sequence exists in which every process can eventually get its maximum and complete.

– If state is unsafe (deadlock could occur), request is denied — process waits.

– $O(n^2)$ time per request; impractical for systems with thousands of processes.

• **Detection and Recovery** — allow deadlock, detect it, recover:

– Detection: scan resource allocation graph for cycles. Cycle = deadlock for single-instance resources.

– Recovery by process termination: kill one or more deadlocked processes (select victim by priority or progress).

– Recovery by rollback: roll process back to checkpoint before it acquired the contested resource. Used in databases (transaction abort and retry).

2.6 Distributed Mutex Algorithms

• In distributed systems: no shared memory, no global clock. All mutual exclusion must be achieved purely through message passing.

• **Lamport's Distributed Mutex (1978)**:

– To enter CS: broadcast timestamped REQUEST to all; enqueue locally; wait for REPLY from all others.

– Receivers: if not wanting CS, immediately REPLY. If wanting/in CS, defer REPLY until after RELEASE.

– After finishing CS: broadcast RELEASE; remove own entry from queue.

– Message cost: 3(N–1) messages per CS entry (request + reply + release to each of N–1 others).

• **Ricart-Agrawala Algorithm**:

– Merges REPLY and RELEASE: deferred replies are sent when CS is released.

– Message cost reduced to 2(N–1) per CS entry — half of Lamport's algorithm.

• **Token Ring Algorithm**:

– Single token circulates around a logical ring. Process may enter CS only when it holds the token.

– After finishing CS, token passed to next process in ring.

– Bounded waiting: at most N–1 token passes before a process gets the token.

– Weakness: lost token (process/link failure) must be detected and regenerated — requires a separate recovery protocol.

• **Two-Phase Commit (2PC)** — for distributed atomic transactions:

– Phase 1 (Prepare): coordinator sends PREPARE to all participants; each logs state and replies VOTE-COMMIT or VOTE-ABORT.

– Phase 2 (Commit): if all replied VOTE-COMMIT, coordinator broadcasts COMMIT; all apply and send ACK. If any VOTE-ABORT, coordinator broadcasts ABORT.

– Blocking problem: if coordinator fails after Phase 1, participants in uncertain state cannot proceed without coordinator recovery. This is the fundamental limitation of 2PC.

– 3PC (Three-Phase Commit) adds a pre-commit phase to eliminate this blocking at the cost of additional messages.

3. Fault Tolerance

Fault tolerance is the ability of a system to continue operating correctly — possibly at reduced performance — despite hardware failures, software bugs, or environmental faults. Achieved through redundancy, error detection, and recovery mechanisms.

3.1 Fault / Error / Failure — Precise Definitions

Term	Precise Definition	Example
Fault	Root cause — physical/logical defect that can potentially cause problems	Bit flip in DRAM from cosmic ray; bug in interrupt handler
Error	Incorrect system state resulting from an activated fault	Wrong value stored in a variable after the bit flip is read
Failure	External deviation from specification visible to users/dependents	Program returns wrong result; service crashes

• Not every fault produces an error — the faulty component may not be exercised during that execution.

• Not every error produces a failure — ECC memory, redundancy, or unused code paths can mask errors before they affect output.

• Goal of fault tolerance: break the fault → error → failure propagation chain at the earliest point.

3.2 Fault Classification

• **Transient faults** — occur once and disappear without any physical change:

– Caused by: cosmic radiation, electromagnetic interference, power supply glitches.

– Most common fault type in modern DRAM systems. ECC DRAM corrects all single-bit transient faults automatically.

– Re-executing the computation on the same hardware will succeed.

• **Intermittent faults** — occur sporadically, extremely difficult to diagnose:

– Caused by: loose connectors, unstable power regulators, temperature-dependent circuit behavior.

– Often precede permanent failures. Detected through statistical analysis of error logs over time.

– Require stress testing, thermal cycling, and long-duration diagnostics.

• **Permanent faults** — persist until repaired or component replaced:

– Examples: burned-out transistor, broken wire, failed disk head, crashed software service.

– Require hardware repair or software reconfiguration (e.g., marking bad disk sectors, disabling a failed DRAM row via row remapping).

• **Byzantine faults** — the most severe class; faulty component sends arbitrary/inconsistent/deceptive messages:

– Arise from: hardware data corruption, software bugs producing plausible-looking wrong outputs, or malicious tampering.

– Standard fail-stop algorithms (assume faulty node simply stops) are incorrect under Byzantine faults.

– Requires completely different, more expensive protocols (PBFT, BFT consensus).

3.3 Redundancy Techniques

• **Triple Modular Redundancy (TMR)** — hardware redundancy:

– Three identical hardware units independently compute the same result.

– A majority voter circuit takes the value agreed upon by at least two of three as the correct output.

– Masks any single-unit failure completely transparently. Used in: aerospace flight computers, nuclear plant controls, spacecraft.

– Cost: 3× hardware + voter circuit; 3× power consumption.

• **RAID (Redundant Array of Independent Disks)**:

– **RAID 0**: Striping only. Best performance; no fault tolerance. Single disk failure = total data loss.

– **RAID 1**: Full mirroring on 2 disks. Tolerates 1 disk failure. Read throughput doubles; write unchanged. 50% storage efficiency.

– **RAID 5**: Striping + distributed parity across N≥3 disks. Tolerates 1 disk failure. Parity distributed so no single parity disk bottleneck. Good read performance; write requires parity update.

– **RAID 6**: Two independent parity calculations across N≥4 disks. Tolerates 2 simultaneous disk failures. Used in large storage arrays where probability of double failure during RAID 5 rebuild is non-negligible.

– **RAID 10**: Stripe of mirrors. Tolerates 1 failure per mirror set. High performance + redundancy. Used in databases.

• **N-Version Programming** — software redundancy:

– N independent implementations of the same specification developed by separate teams.

– Consensus mechanism selects result agreed upon by majority.

– Assumption: independent teams make different mistakes at different points.

– Limitation: studies show independent teams often make the same mistake at the same boundary condition.

• **Error Correcting Codes (ECC)** — information redundancy:

– **Parity bit**: detects single-bit errors; cannot correct.

– **Hamming SEC-DED codes**: add log■(N)+1 check bits to N data bits. Corrects all single-bit errors; detects all double-bit errors.

– **Reed-Solomon codes**: correct multiple symbol errors. Used in: QR codes, CDs/DVDs, deep-space communication, RAID 5/6 parity.

– **CRC-32**: detects burst errors in transmitted data. Used in: Ethernet frames, ZIP files, disk sectors.

• **Time redundancy** — re-execute the computation and compare results:

– If a transient fault caused an error, probability that same transient occurs at the exact same point in re-execution is negligibly small.

– Effective only against transient faults. Permanent faults produce the same error every execution.

3.4 Checkpointing and Rollback Recovery

• **Checkpointing**: saves complete process state (registers, stack, heap, open FDs, network state) to stable non-volatile storage at regular intervals.

• On failure: restart from most recent checkpoint and re-execute, losing only work done since checkpoint.

• Checkpoint interval trade-off:

– Frequent checkpoints: minimize lost work on failure; high I/O bandwidth and storage overhead.

– Infrequent checkpoints: low I/O overhead; more work repeated after each failure.

– Optimal interval: $\tau = \sqrt{(2 \times \text{mean-time-to-failure} \times \text{checkpoint-cost})}$.

• **Coordinated checkpointing**:

– All processes checkpoint simultaneously via two-phase protocol similar to 2PC.

– Ensures collective saved state is consistent — no message recorded as sent without being recorded as received.

– Cost: all processes pause computation during the checkpoint window (global pause).

• **Uncoordinated checkpointing**:

– Each process checkpoints independently at different times. Avoids global pause.

– **Domino effect**: inconsistent checkpoints can cascade rollbacks across processes all the way back to initial state.

– Example: Process A's checkpoint records sending a message that B's checkpoint doesn't record receiving. Rollback of A forces rollback of B to before it received from A, which forces further rollbacks.

• **Message logging** — avoids domino effect:

– Every message received is logged to stable storage before delivery.

– On recovery: restore last checkpoint and replay all logged messages to bring state forward.

– Rollback never needs to go past last checkpoint, regardless of what other processes have done.

– **Pessimistic logging**: log before deliver — safe but adds latency to every message.

– **Optimistic logging**: deliver first, log asynchronously — lower latency but risks losing messages if both process and log host fail.

3.5 Byzantine Fault Tolerance (BFT)

• Byzantine fault: a faulty node may send different values to different recipients, appear internally consistent while being factually wrong, or actively attempt to deceive fault-detection mechanisms.

• **Lamport-Shostak-Pease theorem (1982)**:

– Requires at least **3f+1 total nodes** to tolerate f Byzantine faulty nodes.

– Requires at least **f+1 rounds** of message exchange.

– With f=1 faulty node: minimum 4 nodes required. This is a hard mathematical lower bound — no algorithm can do better.

• **PBFT (Practical BFT) — Castro & Liskov 1999**:

– First practically efficient BFT protocol: $O(n^2)$ message complexity per consensus round.

– Three phases: Pre-prepare (primary sends proposal) → Prepare (replicas broadcast acceptance) → Commit (replicas broadcast commit).

– A replica executes only after receiving 2f+1 matching commit messages — ensures honest majority agreement.

– Used as basis for many blockchain consensus mechanisms.

• BFT applications:

– **Blockchain**: Proof-of-Work achieves BFT-like properties probabilistically; majority honest computation power required.

– **Aerospace**: TMR + BFT protocols in flight control computers where faulty sensor must not override two working ones.

3.6 Availability Metrics

• **MTTF** (Mean Time To Failure): expected operating time until the next failure under normal conditions.

• **MTTR** (Mean Time To Repair): expected time from failure detection to full restoration, including: detection, alert, procurement, replacement, rebuild, software reconfiguration.

• **MTBF** (Mean Time Between Failures) = MTTF + MTTR

• **Availability A = MTTF / (MTTF + MTTR)**

• System with N disks (any disk failure = system failure): system MTTF = disk MTTF / N. Example: 200 disks with individual MTTF = 200,000 hours → system MTTF = 1,000 hours (~6 weeks). Redundancy is not optional at scale.

Nines	Availability	Max downtime/year	Typical use case
3 nines	99.9%	8 h 46 min	Basic web services
4 nines	99.99%	52 min 36 sec	Enterprise applications
5 nines	99.999%	5 min 15 sec	Telecoms, financial systems
6 nines	99.9999%	31.5 seconds	Critical infrastructure
11 nines	99.999999999%	~3 ms	Amazon S3 object durability

4. GPU Architecture and Programming

A GPU is a massively parallel processor with thousands of small cores delivering high throughput for data-parallel computations. Optimized for applying the same operation to millions of data elements simultaneously rather than executing complex sequential logic fast.

4.1 GPU Hardware Architecture

• A modern GPU (NVIDIA H100) contains **132 Streaming Multiprocessors (SMs)**, each housing: 128 CUDA cores, Tensor Cores, a warp scheduler, L1 cache/shared memory (configurable), a large register file, and Special Function Units (SFUs) for transcendentals (sin, cos, sqrt).

• Total CUDA cores: 132 × 128 = **16,896** CUDA cores. Plus 528 Tensor Cores for matrix-multiply-accumulate (used in deep learning).

• GPU memory: **HBM3** (High Bandwidth Memory) mounted directly on the GPU package.

– Bandwidth: 3.35 TB/s — 10× higher than CPU DRAM bandwidth (~300 GB/s DDR5).

– Latency: ~80 ns — higher than CPU L3 cache but hidden by switching between many warps.

– Capacity: 80 GB. Data must be explicitly transferred from CPU DRAM via PCIe (64 GB/s) or NVLink (900 GB/s for H100).

Feature	CPU (Xeon Platinum)	GPU (NVIDIA H100)
Cores	Up to 64 large OoO cores	16,896 small CUDA cores
Clock	2–4 GHz	1.7–2.0 GHz
Peak FP32	~10 TFLOPS	60 TFLOPS
Memory BW	~300 GB/s DDR5	3.35 TB/s HBM3
Memory cap	Up to 4 TB	80 GB
Branch handling	Deep branch prediction + OoO	SIMT; divergent branches serialized
Latency hiding	Out-of-order execution + caches	Warp switching (zero-overhead context switch)
Best workload	Low-latency, complex control flow	High-throughput, regular data-parallel

4.2 CUDA Execution Hierarchy

• **Thread**: smallest execution unit. Has unique threadIdx (within block) and blockIdx (within grid). Computes its data index: global_id = blockIdx.x × blockDim.x + threadIdx.x.

• **Warp**: 32 threads that execute the SAME instruction every cycle (SIMT). The SM's warp scheduler maintains 32–64 resident warps and switches between them in ZERO cycles whenever the active warp stalls on memory.

• **Block**: group of threads (up to 1,024) sharing fast on-chip shared memory and able to synchronize via __syncthreads(). Always runs on ONE SM — never split across SMs.

• **Grid**: complete set of blocks for a single kernel launch. Can be 1D, 2D, or 3D. All blocks of a grid execute the same kernel.

• Kernel launch syntax: **kernel<<<gridDim, blockDim>>>(args)**

– gridDim: number of blocks per dimension. blockDim: number of threads per block per dimension.

– Example: 1,000,000 elements, 256 threads/block → gridDim = ■1000000/256■ = 3907 blocks.

• **GPU Memory Hierarchy**:

– **Registers**: per-thread, fastest (~1 cycle). Private to each thread. ~65,536 32-bit registers per SM shared across all resident threads.

– **Shared Memory**: per-block, on-chip SRAM (~4 ns, ~100× faster than global mem). Manually managed by programmer via __shared__ declaration.

– **L1/L2 Cache**: automatic hardware cache for global memory reads. L1 is per-SM; L2 is chip-wide.

– **Global Memory**: HBM3 off-chip DRAM. Accessible by all threads. Large but ~80 ns latency.

– **Constant Memory**: 64 KB read-only, cached, broadcast to all threads in a warp simultaneously. Used for read-only parameters.

– **Texture Memory**: read-only, 2D spatially cached. Optimized for 2D locality patterns (image processing).

4.3 Warp Execution — SIMT and Branch Divergence

• SIMT (Single Instruction Multiple Threads): all 32 threads in a warp fetch and execute the same instruction every clock cycle, each on its own data.

• Latency hiding via warp switching: when a warp stalls on a memory access (~80 ns = ~160 cycles at 2 GHz), the SM's warp scheduler immediately switches to another ready warp at zero cost. No pipeline flush; no context save. This is how GPUs tolerate high memory latency without large caches.

• **Branch divergence**: when threads within a warp take different if-else paths:

– GPU executes the TRUE branch first with threads that took it (masking off the others with a predicate mask).

– Then executes the FALSE branch with the remaining threads (masking off the first group).

– Both paths execute sequentially — potentially doubling execution time for a 50/50 split.

– Only affects threads within the SAME warp. Different warps can take different paths without any penalty.

• Minimizing divergence: organize data so adjacent threads (sharing a warp) always take the same branch. Sort data before kernel launch if needed.

4.4 GPU Performance Optimization Techniques

• **Coalesced memory access** — most impactful single optimization:

– When 32 threads in a warp access 32 consecutive 4-byte floats at a 128-byte-aligned address, ALL 32 accesses are served in ONE 128-byte memory transaction.

– Strided access (thread i accesses address i×stride): each thread may need its own transaction → up to 32× lower effective bandwidth.

– Solution: use Structure of Arrays (SoA) layout instead of Array of Structures (AoS) to ensure column-wise accesses are contiguous.

• **Shared memory tiling**:

– Load a tile (block) of data from slow global memory into fast shared memory cooperatively (all threads in block contribute).

– All computation within the tile runs from shared memory (~4 ns) instead of global memory (~80 ns).

– For matrix multiplication: reduces global memory accesses from $O(N^3)$ to $O(N^3/\text{tile_size})$, often achieving 10–30× speedup.

• **Bank conflicts in shared memory**:

– Shared memory is divided into 32 banks of 4 bytes each. Optimal: each thread in warp accesses a different bank.

– Conflict: multiple threads access different addresses in the SAME bank → serialized N-way → N× throughput reduction.

– Fix: pad shared arrays. Use float tile[32][**33**] instead of [32][32] so column accesses span different banks.

• **Occupancy** — ratio of active warps to maximum warps per SM:

– Limited by: register usage per thread (more registers → fewer threads fit → lower occupancy), shared memory per block, and block size.

– Higher occupancy → more warps to hide memory latency. Target ≥50% occupancy as a rule of thumb.

– Use NVIDIA Nsight Compute to measure achieved vs theoretical occupancy and identify the limiting resource.

• **CUDA Streams** — overlap kernel execution with data transfer:

– Default stream: data transfer and kernel execution are serialized.

– Multiple streams: while GPU processes batch N on stream 0, CPU transfers batch N+1 to GPU on stream 1. True overlap.

– Required for fully utilizing PCIe bandwidth alongside GPU compute.

5. Heterogeneity in Distributed Systems

Heterogeneity is the coexistence of diverse hardware, operating systems, programming languages, communication protocols, and data representations within one distributed system. Managing it is a fundamental distributed systems engineering challenge.

5.1 Dimensions of Heterogeneity

• **Network heterogeneity**: different parts of a system connected by networks with vastly different characteristics.

– Intra-rack HPC links: InfiniBand HDR at 200 Gb/s, sub-microsecond latency.

– Inter-datacenter WAN links: 10–100 Gb/s, 20–200 ms round-trip latency.

– Algorithms optimal for low-latency networks (many small messages) perform poorly over high-latency WANs, where batching is essential.

• **Hardware heterogeneity**: different CPU ISAs, memory capacities, accelerator types.

– CPU ISAs: x86-64 (Intel/AMD), ARM64 (Apple M-series, AWS Graviton), RISC-V, PowerPC.

– Accelerators in same cluster: NVIDIA A100, AMD MI250X, Google TPU, Intel Habana Gaudi.

– Frameworks like PyTorch/TensorFlow provide hardware abstraction layers hiding these differences through unified operator libraries.

• **OS heterogeneity**: nodes running different operating systems simultaneously.

– HPC clusters typically standardize on Linux; cloud environments mix Linux and Windows.

– Syscall interfaces differ; file paths differ (/ vs \); endianness may differ.

• **Data representation heterogeneity**:

– **Endianness**: x86 = little-endian (LSB at lowest address); SPARC/historical MIPS = big-endian. Network byte order is big-endian; POSIX provides htons/htonl/ntohs/ntohl for conversion.

– **String encoding**: UTF-8 vs UTF-16 vs ASCII; null-terminated vs length-prefixed.

– **Floating-point**: all modern systems use IEEE 754, but historical non-IEEE systems exist in legacy environments.

• **Programming language heterogeneity**:

– Different components written in C++, Python, Java, Fortran, Rust.

– Each language boundary requires marshalling/unmarshalling (data conversion between type systems and memory layouts).

– **Protocol Buffers (protobuf)** and **Apache Thrift**: define IDL (Interface Description Language); auto-generate serialization code in multiple languages.

5.2 Middleware and OpenCL

• Middleware is a software layer between the OS and distributed applications providing uniform high-level abstractions that hide heterogeneity.

• Types of middleware:

– **RPC / gRPC**: makes cross-network calls look like local function calls. Client stub marshals args; server skeleton unmarshals; response returned transparently. gRPC uses Protocol Buffers + HTTP/2, providing streaming, TLS, bi-directional communication.

– **Message Passing**: MPI (HPC), ZeroMQ, RabbitMQ, Kafka. Explicit message exchange; decouples senders and receivers in time.

– **Object Invocation**: CORBA, Java RMI. Invoke methods on remote objects as if local. Middleware handles object location, parameter marshalling, and network transport.

– **Web Services**: REST (HTTP + JSON/XML) and SOAP. Language-neutral; firewall-friendly; widely used in cloud APIs.

• **OpenCL** (Open Computing Language):

– Open standard for heterogeneous parallel programming across CPUs, GPUs, FPGAs, DSPs from ANY vendor.

– Host code (CPU) manages memory and kernel launches. Kernel code compiled at runtime by device driver to native machine code for target architecture.

– Same kernel source runs on NVIDIA GPU, AMD GPU, Intel CPU, or FPGA without source changes.

– Performance gap vs native CUDA: OpenCL on NVIDIA hardware typically achieves 80–95% of equivalent CUDA performance due to compiler optimization differences.

6. Interconnection Topologies

Interconnection topology defines the physical or logical arrangement of nodes and links in a parallel system. It directly determines worst-case communication latency (diameter), maximum collective bandwidth (bisection width), hardware wiring cost (degree), and fault tolerance.

6.1 Key Metrics

• **Diameter**: maximum shortest-path hop count between any two nodes. Bounds worst-case message latency. Smaller = better.

• **Bisection width**: minimum number of links to cut the network into two equal halves. Bottleneck bandwidth for all-to-all communication (FFT, matrix transpose, deep learning). Larger = better.

• **Node degree**: number of links per node. Higher degree = more wiring and router ports = higher hardware cost and power. Must be bounded for practical scalability.

• **Connectivity**: minimum number of independent paths between any two nodes. Determines fault tolerance — connectivity k means system remains connected after k–1 link failures.

• **Symmetry**: all nodes are structurally equivalent. Simplifies routing algorithms and load balancing; all nodes have same worst-case communication distance.

6.2 Topology Comparison Table

Topology	N Nodes	Diameter	Degree	Bisection W.	Used In
Linear Array	N	N–1	2	1	Systolic arrays, rarely standalone
Ring	N	N/2	2	2	Token ring, systolic, on-chip
2D Mesh	N	2(√N–1)	≤4	√N	Early Cray, on-chip NoCs
2D Torus	N	√N	4	2√N	IBM BlueGene/L, on-chip
3D Torus	N	3N<super>1/3</super>/2	6	N<super>2/3</super>	IBM BlueGene/Q, Cray XT5/XK7
Hypercube	2<super>n</super>	n=log■N	n=log■N	N/2	Intel iPSC (historical)
Fat Tree	N	2log(N/k)	variable	N/2	InfiniBand HPC (dominant today)
Dragonfly	N	3	high fixed	~N/2	HPE Slingshot, Cray Aries, Frontier

6.3 Hypercube — Detailed Analysis

• An **n-dimensional hypercube** has exactly 2^n nodes. Each node labeled with unique n-bit binary string.

• Link rule: two nodes are connected if and only if their binary labels differ in exactly ONE bit position (Hamming distance 1).

  – Example (n=4): node 0110 is directly connected to 0111, 0100, 0010, 1110 (flip each of the 4 bit positions).

  – Each node therefore has exactly n neighbors — one per bit position.

• Properties:

  – Diameter = n = log■N. Any message reaches any destination in at most log■N hops.

  – Bisection width = N/2 — optimal for degree n. Maximum possible given the wiring constraint.

  – Connectivity = n. System remains connected after up to n–1 simultaneous link failures.

• **E-cube routing** (dimension-order routing): route by correcting differing bits one dimension at a time.

– Guaranteed deadlock-free shortest-path routing. Hardware implementation is simple.

• Scalability limitation: node degree grows as log■N.

– 1,024 nodes (n=10): degree 10. 1,048,576 nodes (n=20): degree 20.

– Modern routers have fixed port counts (32–64 ports). Hypercubes become impractical beyond a few thousand nodes.

– Fat trees and Dragonfly networks replaced hypercubes by achieving similar diameter with **bounded** node degree.

6.4 Fat Tree — Modern HPC Standard

• Fat tree solves the root bottleneck of simple tree networks: bandwidth increases toward the root so no single link becomes a bottleneck.

• Implementation: **folded Clos network** — three-tier switching hierarchy.

– **Edge switches**: connect directly to compute nodes.

– **Aggregation switches**: connect edge switches within a pod.

– **Core switches**: connect pods.

• With k-port switches: supports $k^3/4$ nodes with **full bisection bandwidth** — every node can communicate with every other at full line rate simultaneously.

• InfiniBand fat tree clusters: 70% of TOP500 supercomputers use this topology. TACC Frontera (~8,000 nodes), NERSC Perlmutter (~6,000 GPU nodes).

• **Dragonfly topology** (newest generation):

– High-radix routers (64+ ports) connected all-to-all within a group; sparse connections between groups.

– Diameter of 3 hops regardless of system size. HPE Slingshot (Frontier, 9,408 nodes) and Cray Aries (Titan) use this.

7. Load Balancing

Load balancing distributes computational work across processors so all remain productively busy throughout execution, minimizing total completion time by eliminating idle time caused by uneven work distribution.

7.1 Static Load Balancing

• Work partitioned before execution begins; assignments fixed for entire computation. No runtime overhead.

• **Block partitioning**: process i handles iterations [i×N/P, (i+1)×N/P).

– Perfect balance when all iterations have equal cost. Used for dense matrix operations.

– Fails for triangular matrices: process 0 handles top rows (little work), process P–1 handles bottom rows (much more work). Severe imbalance.

• **Cyclic partitioning**: iteration i assigned to process (i mod P), round-robin.

– Naturally balances triangular work: early (small) and late (large) iterations interleaved across all processes.

– Downside: stride P access pattern destroys cache spatial locality.

• **Block-cyclic partitioning**: assign blocks of B consecutive iterations cyclically.

– Used in ScaLAPACK for distributed dense linear algebra. Balances load while partially preserving locality.

– Tuning B: small B = good balance, bad locality. Large B = good locality, potentially bad balance.

• **Graph partitioning**: for computations defined on irregular graphs (FEM meshes, sparse matrices, social networks).

– Goal: partition vertices into P equal-sized subsets minimizing cut edges (cut edges = inter-process communication).

– NP-hard in general but near-optimal solutions computed in O(N) time by multilevel methods: METIS, ParMETIS, Scotch.

– Multilevel approach: coarsen graph → partition small graph → uncoarsen with local refinement at each level.

7.2 Dynamic Load Balancing

• Work redistributed at runtime based on actual load. Essential for irregular problems with unpredictable task sizes.

• **Master-Worker (Task Farm)**:

– Master maintains queue of tasks. Workers request tasks, execute, return results, request next task.

– Naturally balances: fast workers complete more tasks, slow workers fewer. No idle time in principle.

– Bottleneck: master becomes a hotspot at thousands of workers or for microsecond-duration tasks.

– Solution: hierarchical task farming — multiple levels of masters to distribute the bottleneck.

• **Work Stealing** (most scalable dynamic approach):

– Each processor has a double-ended queue (deque) of tasks. Adds/removes from its own local end (no synchronization needed most of the time).

– When a processor's deque is empty: randomly selects a victim and steals tasks from the victim's remote end.

– Randomized victim selection prevents hotspots. Stealing from remote end avoids contention with victim's local operations.

– Cilk's work-stealing scheduler proves: expected runtime = W/P + O(D) where W=total work, D=critical path depth. Near-optimal.

• **Self-Scheduling** (for parallel loops):

– **Dynamic/chunk=1**: each idle processor takes exactly one next iteration. Perfect balance; high overhead per task.

– **Guided scheduling**: starts with large chunks, exponentially decreasing to 1. Reduces overhead while achieving good balance.

– **Adaptive/Work-Queue**: chunk size chosen based on remaining iterations and processor count.

7.3 Metrics and Measurement

• **Load Imbalance Factor** = T_max / T_avg — ratio of maximum to average execution time across processors.

• **Parallel efficiency** = T_avg / T_max × 100%. Perfect balance: T_max = T_avg, efficiency = 100%.

• Measurement: use profiling tools (TAU, Score-P) to plot per-process execution time. Gaps in the timeline visualization show idle time from load imbalance.

8. Memory Consistency Models

A memory consistency model is a formal contract specifying which values a read operation may return, given the history of preceding writes. It defines the ordering guarantees the hardware and compiler provide to the programmer regarding memory operation visibility across threads.

8.1 Sequential Consistency (SC)

• Defined by Lamport (1979): result of any execution is the same as if all memory operations of all processors executed in some single sequential order, and each processor's operations appear in that sequence in program order.

• Intuitive model: the execution looks like a single global interleaving of all threads' operations. No operation can appear to 'go back in time' relative to another processor's view.

• SC is expensive to implement: prevents all hardware reordering optimizations (store buffers, out-of-order execution, speculative reads). Inserting a full fence after every load and store adds 10–30% overhead. No modern processor implements SC natively.

• Programmers who reason informally about shared-memory programs are usually implicitly assuming SC. Understanding relaxed models explains why programs correct under SC assumptions can fail on real hardware.

8.2 Relaxed Consistency Models

• **Total Store Order (TSO)** — x86/x64 Intel and AMD:

– Relaxes one SC rule: stores from a processor may be delayed in its store buffer before becoming globally visible.

– A processor's own subsequent loads can execute before an earlier store has propagated to other processors.

– Result: a processor may read a stale value written by another processor that logically should have seen this processor's earlier store.

– Requires explicit MFENCE or LOCK prefix to prevent store-buffer bypassing in critical synchronization code.

• **Release Consistency** — ARM, PowerPC:

– Only synchronization operations (acquire and release) enforce ordering. Between syncs, hardware may reorder any load/store with any other freely.

– **Acquire** load: all subsequent loads/stores must not execute before the acquire completes.

– **Release** store: all preceding loads/stores must complete before the release becomes visible.

– Gives hardware maximum freedom to optimize; requires careful programmer annotation of every critical section and inter-thread communication.

• **C++11 / Java Memory Model**:

– Defined by **happens-before** relation. Write W happens-before read R if connected by program order within one thread OR by a synchronization edge (mutex unlock → lock, atomic store → load).

– A read is guaranteed to see the most recent write that happens-before it. Operations with no happens-before relationship are concurrent → data race → undefined behavior in C++11.

8.3 Cache Coherence — MESI Protocol

• Cache coherence: guarantees all processors see a consistent view of memory despite each having a private cache. Without it, one processor could read a stale cached value after another has updated it.

• **MESI protocol** — 4 states per cache line per cache:

– **Modified (M)**: present only in this cache; modified (differs from memory). Must write back before any other cache reads it.

– **Exclusive (E)**: present only in this cache; clean (matches memory). Can be written without broadcasting if no other cache holds it — fast upgrade to M.

– **Shared (S)**: may be present in multiple caches; clean. Any read allowed; write requires invalidating all other copies first.

– **Invalid (I)**: line not present in this cache or outdated; must be fetched on next access.

• **Write to a Shared line — Read for Ownership (RFO)**:

– Send Invalidate message to all caches holding the line via bus or directory.

– Wait for Invalidation Acknowledgments from all holders.

– Perform write; transition to Modified state.

– RFO round-trip is the primary coherence overhead in shared-memory multiprocessors.

• **False sharing** — a major hidden performance problem:

– Two processors frequently write to DIFFERENT variables that share the SAME 64-byte cache line.

– Each write invalidates the other processor's cached copy, generating constant RFO traffic even though the variables are logically independent.

– Symptom: adding more threads makes the program SLOWER (super-linear slowdown).

– Fix: pad structures to align each thread's data to a separate cache line — use **alignas(64)** in C++17.

• **Directory-based coherence** — for large-scale NUMA systems (no shared bus):

– Directory entry per memory line tracks which processor caches hold a copy.

– Invalidations sent point-to-point via directory instead of broadcast — scales to thousands of processors.

– Used in: AMD EPYC (Infinity Fabric), Intel Xeon Scalable (UPI mesh), IBM POWER.

9. Memory Hierarchies

Memory hierarchy organizes storage technologies by speed, capacity, and cost. Fast technologies are small and expensive; slow technologies are large and cheap. Exploiting locality of reference makes programs behave as if they have fast large memory.

9.1 Full Hierarchy with Numbers

Level	Technology	Size	Latency	Bandwidth	Managed By
Registers	Flip-flops	< 1 KB	< 0.3 ns (1 cycle)	Unlimited	Compiler/CPU
L1 Cache	SRAM	32–64 KB	~1–4 ns	~3 TB/s	Hardware
L2 Cache	SRAM	256 KB–2 MB	~3–10 ns	~1 TB/s	Hardware
L3 Cache	SRAM	4–64 MB	~10–30 ns	~500 GB/s	Hardware
Main Memory	DRAM	4 GB–4 TB	~60–100 ns	50–300 GB/s	OS
NVMe SSD	NAND Flash	1–30 TB	~100–200 µs	~7 GB/s	OS
SATA SSD	NAND Flash	1–16 TB	~200–500 µs	~550 MB/s	OS
HDD	Magnetic disk	1–20 TB	~5–10 ms	~200 MB/s	OS
Network Storage	Remote	Unlimited	1 ms – 1 s	~1–100 Gb/s	Distributed FS

9.2 Locality of Reference

• **Temporal locality**: a memory location accessed at time t is likely accessed again soon. Exploited by caches: on first access, data is fetched into fast cache; subsequent accesses find it there.

– Exploited by: caches keeping recently used lines, loop variables held in registers, frequently called functions cached in L1I.

• **Spatial locality**: if address A is accessed, nearby addresses are likely accessed soon. Exploited by fetching an entire 64-byte cache line on each miss, not just the requested word.

– Array traversal (array[0], array[1], ..., array[N]): perfect spatial locality — cache line prefetches next 15 floats for free on each miss.

– Linked list traversal: poor spatial locality — nodes scattered throughout memory; near-zero benefit from prefetching.

• **Hardware prefetchers**: detect regular access patterns and prefetch data before it's requested.

– Next-line prefetcher: always prefetches cache line N+1 after accessing line N.

– Stride prefetcher: detects constant-stride patterns (e.g., every 64 bytes for a column of a row-major matrix).

– Stream prefetcher: detects multiple simultaneous sequential access streams from different base addresses.

– Limitation: pointer-chasing patterns (linked lists, tree traversal) are unpredictable — next address depends on current data load.

9.3 Cache Performance

• **AMAT** (Average Memory Access Time) = Hit_time + Miss_rate × Miss_penalty

– Example: L1 hit time = 4 cycles, L1 miss rate = 5%, L2 hit time = 10 cycles, L2 miss rate = 1%, memory latency = 200 cycles.

– AMAT = 4 + 0.05 × (10 + 0.01 × 200) = 4 + 0.05 × 12 = 4.6 cycles average.

• **Three Cs of cache misses**:

– **Compulsory (Cold)**: first-ever access to a block. Unavoidable; can be reduced by prefetching.

– **Capacity**: working set larger than cache size. Reduced by loop blocking/tiling to fit working set in cache.

– **Conflict**: multiple addresses map to same set in a set-associative cache. Reduced by higher associativity or array padding.

9.4 NUMA Architecture

• NUMA (Non-Uniform Memory Access): in multi-socket servers, each socket has its own local DRAM. A core accesses local DRAM fast and remote (other socket's) DRAM slower.

• Typical NUMA penalty: local access ~60 ns; remote access ~120 ns (2× ratio). Cross-socket traffic uses QPI/UPI interconnect at finite bandwidth (~100 GB/s).

• NUMA-aware programming:

– **numactl --membind=0**: bind process memory to NUMA node 0.

– **numactl --cpunodebind=0**: bind process threads to cores on node 0.

– **libnuma numa_alloc_onnode(size, node)**: allocate memory on specific NUMA node programmatically.

• **First-touch policy** (Linux default): memory page allocated on the NUMA node of the first thread to write to it.

– Problem: if main thread initializes all data before spawning workers, ALL data sits on main thread's NUMA node. All workers on other nodes suffer 2× latency.

– Fix: initialize data in parallel with the SAME thread affinity used for computation. Worker thread touches its own data partition during initialization.

10. Message Passing Interface (MPI)

MPI is the de facto standard library for parallel programming on distributed-memory systems, providing portable point-to-point and collective communication for programs running on anything from a laptop to a 100,000-node supercomputer without source-code changes.

10.1 Execution Model

• SPMD (Single Program Multiple Data): all P processes run the same program binary but follow different code paths based on their unique **rank** (integer 0 to P–1).

• Processes communicate ONLY through messages — no shared memory. All data exchange must be explicit.

• **Communicator**: defines a communication domain. MPI_COMM_WORLD = all P processes.

– MPI_Comm_split: creates sub-communicators (e.g., all processes in same matrix row).

– MPI_Cart_create: creates communicator with virtual Cartesian topology for neighbor-aware operations.

– Context tag in communicator ensures messages from different communicators never accidentally match.

• Message matching: based on 4-tuple (source, destination, tag, communicator).

– MPI_ANY_SOURCE and MPI_ANY_TAG wildcards available in receive.

– Messages from a specific source to specific destination are received in FIFO order — guaranteed by MPI standard.

10.2 Send Modes — All Six

Mode	Send Call	Blocking?	Completes When	Use Case
Standard	MPI_Send	Yes	Buffer safe to reuse (may internally buffer)	Default
Synchronous	MPI_Ssend	Yes	Receiver has posted matching MPI_Recv	Confirm receiver readiness
Buffered	MPI_Bsend	Yes (fast)	Data copied to user-provided buffer	Avoid blocking without async complexity
Ready	MPI_Rsend	Yes	Sent (receiver MUST have already posted recv)	Micro-optimization; dangerous if misused
Non-blocking Std	MPI_Isend	No (returns immediately)	Must call MPI_Wait before reusing buffer	Overlap comm with compute
Persistent	MPI_Send_init + MPI_Start	No	Must call MPI_Wait	Repeated sends same dest/type; lowest

10.3 Collective Operations — All Types

Operation	MPI Function	Communication Pattern	Cost (Hockney model)
Broadcast	MPI_Bcast	Root → all	O(log P × (α + β×n))
Scatter	MPI_Scatter	Root → each (distinct chunks)	O(log P × α + β×n)
Gather	MPI_Gather	Each → root (assemble)	O(log P × α + β×n)
Allgather	MPI_Allgather	Each → all (all get full data)	O(log P × (α + β×n))
Reduce	MPI_Reduce	All → root (with operator)	O(log P × (α + β×n))
Allreduce	MPI_Allreduce	All → all (reduce + broadcast)	O(2 log P × (α + β×n))
Reduce-Scatter	MPI_Reduce_scatter	Reduce then scatter result	O(log P × (α + β×n))

Scan (prefix)	MPI_Scan	Prefix reduction: proc i gets result for 0..i	O(log P × (α + β×n))
Alltoall	MPI_Alltoall	Complete exchange: each → each	O(P×α + β×n) ← most expensive
Barrier	MPI_Barrier	Synchronize all processes	O(2 log P × α)

• α = per-message startup latency (µs). β = per-byte transfer cost (ns/byte). n = message size (bytes).

• Alltoall is the most expensive collective — O(P×α) latency grows linearly with P. It is the bottleneck in parallel FFT, matrix transposition, and all-to-all ML communication.

10.4 MPI Derived Datatypes and Topologies

• Derived datatypes allow sending non-contiguous memory in a single MPI message without manual packing.

– **MPI_Type_vector(count, blocklength, stride, oldtype, &newtype;)**: describes 'count' blocks of 'blocklength' elements separated by 'stride' — e.g., a column of a row-major 2D array.

– **MPI_Type_struct**: describes a heterogeneous struct with mixed data types at given offsets.

– **MPI_Type_commit(&newtype;)**: must commit before using. **MPI_Type_free(&newtype;)**: release when done.

• **Cartesian topology (MPI_Cart_create)**:

– Maps ranks to coordinates in an N-dimensional grid. MPI_Cart_shift(comm, dim, disp, &src;, &dest;) finds neighbor ranks.

– Example: 16 processes in 4×4 grid. Process (2,3) calls MPI_Cart_shift for dim=0,disp=1 to find rank below it in the grid.

11. MIMD and SIMD Architectures

Flynn's Taxonomy (1966) classifies processors by the number of simultaneous instruction and data streams. SIMD applies one instruction to many data elements simultaneously; MIMD runs independent instruction streams on independent data on multiple processors.

11.1 Flynn's Full Taxonomy

Class	Instructions	Data Streams	Description	Real Examples
SISD	Single	Single	Classic sequential processor; one instruction at a time	Uniprocessor (historical)
SIMD	Single	Multiple	Same instruction applied to a vector of data elements	SSE/AVX, GPU warps, DSPs
MISD	Multiple	Single	Multiple operations on same data stream	Fault-tolerant pipelines (rare)
MIMD	Multiple	Multiple	Each processor executes own instruction stream	Clusters, SMP servers, multicores

11.2 SIMD — Vector Processing in Depth

• SIMD applies the same operation to multiple data elements simultaneously using wide vector registers, exploiting data-level parallelism within a single processor.

• **x86 SIMD ISA evolution**:

– **MMX** (1996): 64-bit registers, integer only.

– **SSE/SSE2** (1999/2001): 128-bit XMM registers; 4 floats or 2 doubles per instruction.

– **AVX** (2011): 256-bit YMM registers; 8 floats or 4 doubles per instruction.

– **AVX-512** (2016): 512-bit ZMM registers; 16 floats or 8 doubles per instruction. Up to 16× scalar throughput.

• **Auto-vectorization** by compiler: transforms scalar loops to SIMD instructions automatically.

– Requirements: no loop-carried dependencies, constant-stride memory access, known-bounded iterations, no aliasing.

– Help the compiler: use **restrict** keyword (no aliasing), **__builtin_assume_aligned** (alignment), **#pragma omp simd**.

• **SIMD intrinsics**: direct access to SIMD instructions from C/C++ without assembly.

– **_mm512_add_ps(a, b)**: adds 16 floats using VADDPS zmm instruction.

– **_mm512_mask_add_ps(src, mask, a, b)**: adds only elements where mask bit = 1 (predicated execution for loop tails and conditionals).

• **Alignment requirement**: SIMD loads/stores are most efficient when data is aligned to SIMD register width (32 bytes for AVX, 64 bytes for AVX-512). Misaligned access requires two cache line loads.

11.3 MIMD — Shared vs Distributed Memory

• MIMD: each processor executes its own instruction stream on its own data. Two fundamental sub-classes:

• **Shared-Memory MIMD**:

– All processors share a single physical address space. Communication via shared variables; synchronization via locks/barriers.

– **SMP (Symmetric Multiprocessing)**: uniform memory access — all processors equal distance from all memory. Scales to ~32 cores before memory bus saturates.

– **NUMA**: non-uniform access — each processor has local memory; remote access slower. Scales to ~1,000 cores in a single OS image.

– Programming: OpenMP, pthreads, C++11 threads.

• **Distributed-Memory MIMD**:

– Each processor has private memory; no hardware support for accessing remote memory. Communication only through explicit messages.

– Scales to 100,000+ processors — adding nodes increases both compute and memory proportionally.

– Programming: MPI. All data distribution and communication must be explicit.

Aspect	Shared-Memory MIMD	Distributed-Memory MIMD
Communication	Implicit (shared variables)	Explicit (MPI send/receive)
Programming effort	Lower — OpenMP	Higher — MPI + data decomposition
Scalability	Hundreds of cores max	100,000+ cores/nodes
Memory capacity	Bounded by one system	Scales linearly with node count
Fault isolation	Low — one crash affects all	High — node failures isolated
Hardware	Shared memory bus/NoC	High-speed interconnect (IB, Slingshot)

12. Multithreaded Programming

Multithreaded programming enables concurrent execution of multiple flows of control within a single process, sharing the address space while maintaining independent program counters, registers, and stacks. The primary mechanism for exploiting shared-memory multicore parallelism.

12.1 Thread vs Process

Aspect	Process	Thread
Memory	Own private address space	Shares process heap, globals, code
Stack	Own stack	Own stack; shares everything else
Creation cost	Expensive (copy address space, fork)	Cheap (create stack + TCB)
Context switch	Heavy (~1,000–5,000 cycles + TLB flush)	Light (~100–500 cycles; no TLB flush)
Communication	IPC (pipes, sockets, shared memory)	Direct via shared variables
Crash impact	Isolated — other processes unaffected	Can crash entire process
Use case	Isolation, security, separate apps	Parallelism within one application

12.2 OpenMP — Complete Directive Reference

• **#pragma omp parallel**: creates a team of threads; each executes the block. Variables declared before the region are shared by default; variables declared inside are private.

– Number of threads: set by OMP_NUM_THREADS env variable, omp_set_num_threads(), or num_threads(N) clause.

– Default shared-everything is a common bug source: loop counters and temporaries must be declared private or listed in the **private(var)** clause.

• **#pragma omp parallel for**: distributes loop iterations across threads.

– **schedule(static)**: divide iterations into P equal chunks at compile time. Fast; may be imbalanced for irregular work.

– **schedule(static, chunk)**: assign exactly 'chunk' iterations cyclically. Good balance for moderately irregular work.

– **schedule(dynamic, chunk)**: assign chunks on demand from a work queue. Perfect balance; higher per-task overhead (~1–10 µs per task pickup).

– **schedule(guided)**: start with large chunks, decrease exponentially. Reduces overhead while achieving good balance.

– **schedule(runtime)**: defer decision to OMP_SCHEDULE environment variable.

• **#pragma omp reduction(op:var)**:

– Each thread gets a private copy of 'var' initialized to identity (0 for +, 1 for ×, INT_MIN for max).

– At end of parallel region, all private copies combined using 'op' into the original shared variable.

– Dramatically faster than using a critical section or atomic around the accumulation.

• **#pragma omp critical**: mutual exclusion — only one thread executes the block at a time. High overhead for frequently-entered regions.

• **#pragma omp atomic**: atomic update of a single variable (read-modify-write). Much faster than critical for simple operations like sum += x.

- **#pragma omp task**: creates an asynchronous task executable by any thread.
  - Enables irregular parallelism: recursive divide-and-conquer algorithms (quicksort, tree traversal) expressed naturally.
  - **depend(in:x, out:y)** clause: specifies data dependencies; runtime respects them automatically.
  - **#pragma omp taskwait**: wait for all tasks created in current scope to complete.
- **#pragma omp target** (OpenMP 4.x+): offload region to accelerator (GPU).
  - **map(to: array[0:n])**: copy to device before kernel. **map(from:)**: copy back after. **map(tofrom:)**: both.
  - **teams** and **distribute** constructs distribute work across GPU compute units.

12.3 Common Threading Bugs

- **Data race**: two threads access same variable concurrently; at least one is a write; no synchronization.
  - Behavior in C++11: undefined. Compiler may produce any output.
  - Fix: use std::atomic for simple variables; std::mutex + std::unique_lock for compound operations.
  - Detection: ThreadSanitizer (TSan) — add -fsanitize=thread to GCC/Clang compile flags. ~5–10× overhead.
- **Deadlock**: threads hold locks that each other needs.
  - Classic scenario: Thread A holds lock1, waits for lock2. Thread B holds lock2, waits for lock1.
  - Prevention: always acquire multiple locks in the same global order across all threads.
  - Detection: Helgrind (Valgrind) tracks lock acquisition order graphs at runtime.
- **False sharing**:
  - Two threads write to different variables sharing one 64-byte cache line.
  - Each write invalidates the other's cache copy — constant RFO bus traffic on logically independent data.
  - Symptom: 4-thread program slower than 2-thread program.
  - Fix: **alignas(64) float threadData[N_THREADS]** — pad to separate cache lines.
- **Priority inversion**: high-priority thread waits for a lock held by low-priority thread preempted by a medium-priority thread.
  - Fix: **priority inheritance** — low-priority thread temporarily inherits high-priority thread's priority while holding the contested lock.

13. Parallel Algorithms & Architectures

Parallel algorithms decompose problems into concurrent subtasks, specifying their coordination to reduce runtime. Analysis uses work (total operations) and depth (critical path length) to characterize the fundamental parallelism available.

13.1 Performance Laws

- **Amdahl's Law**: S(P) = 1 / (f + (1–f)/P), where f = serial fraction.
  - – Maximum speedup: as P→∞, S→1/f. Serial fraction f=10% caps speedup at 10×, forever, regardless of how many processors are added.
  - – Serial fraction f=5% caps at 20×. f=1% caps at 100×. Reducing f is the most impactful optimization.
  - – Pessimistic: assumes fixed problem size. Criticized because real scientists solve LARGER problems on more processors.
- **Gustafson's Law**: S_scaled(P) = P – f(P–1), where f = serial fraction at the scaled problem size.
  - – As problem size grows with P, the serial fraction (I/O setup, initialization) shrinks as a fraction of total work.
  - – Example: P=64, f=0.01 (1% serial at that scale): S = 64 – 0.01×63 = 63.37×. Near-linear speedup is achievable.
  - – Better explains why 100,000-core supercomputers deliver value for petascale scientific simulations.
- **Work-Depth model**:
  - – W = total work (equivalent to serial algorithm complexity). D = depth = critical path length.
  - – Lower bound: T(P) ≥ max(W/P, D). No parallel schedule can beat either bound.
  - – Average parallelism = W/D: number of processors that can be usefully employed on average.
  - – Example: parallel sort, W=O(N log N), D=O(log² N). Parallelism = N/log N — can scale to N/log N processors.

13.2 Foster's Parallel Algorithm Design Methodology

- **Step 1 — Partitioning**: identify the finest-grain tasks.
  - – **Domain decomposition**: partition the data; each task operates on its data subset. Used in: stencil codes, matrix ops, N-body simulations.
  - – **Functional decomposition**: partition the computation into distinct functions that can run concurrently. Used in: pipeline parallelism, pipeline stages in streaming.
  - – Goal: maximize exposed parallelism; avoid premature aggregation.
- **Step 2 — Communication**: identify data sharing between tasks.
  - – **Local communication**: each task communicates only with neighbors (stencil, mesh). Low total volume.
  - – **Global communication**: every task communicates with every other (all-to-all, FFT). High total volume.
- **Step 3 — Agglomeration**: group fine-grained tasks into coarser ones.
  - – **Coarse-grained**: fewer, larger tasks. Lower communication overhead; less load balancing flexibility.
  - – **Fine-grained**: many small tasks. Better load balance; higher overhead per task.
  - – Target: task computation time >> communication overhead per task.
- **Step 4 — Mapping**: assign tasks to processors.
  - – Minimize inter-processor communication: co-locate tasks that communicate frequently.

– Maximize processor utilization: balance work across processors.

– Map to match topology: 2D mesh task grid → 2D processor torus reduces communication distance.

13.3 Parallel Reduction and Prefix Scan

• Reduction: compute A[0] ⊕ A[1] ⊕ ... ⊕ A[N–1] for associative operator ⊕ (sum, max, AND, XOR, etc.).

– Tree-based reduction: log■N rounds. Round k: each active pair (at distance 2^{k-1}) reduces to one value. Work = O(N), Depth = O(log N).

– OpenMP: **#pragma omp parallel for reduction(+:sum)** — compiler generates tree reduction automatically.

– CUDA: each block does shared-memory tree reduction; host sums block results. Achieves near-peak memory bandwidth.

• Parallel prefix scan (Blelloch 1990):

– Given A[0..N–1], compute B[i] = A[0] ⊕ ... ⊕ A[i] for all i simultaneously.

– **Up-sweep** (reduce phase): log■N rounds, each element combines with left neighbor at distance 2^k.

– **Down-sweep** (distribution phase): log■N rounds, distribute partial results downward.

– Result: O(N) work, O(log N) depth — optimal.

– Applications of scan: parallel quicksort partition indices, stream compaction, histogram, BFS, radix sort bucket offsets.

13.4 Parallel Sorting

• **Bitonic Sort**: comparison network sorting N (power of 2) elements in log²N phases.

– All comparisons predetermined — no data-dependent branching. All processors know their comparison partner at each step.

– Work: O(N log²N). Depth: O(log²N). Not work-optimal but maps perfectly to GPU warp-uniform execution.

– Used in: GPU sorting (CUDA thrust::sort), butterfly network hardware sorters.

• **Parallel Sample Sort** (best for distributed memory):

– Phase 1: each process sorts its local N/P data.

– Phase 2: each process samples s elements; root sorts all samples, selects P–1 splitters, broadcasts.

– Phase 3: each process binary-searches splitters to bucket its data; all-to-all exchange sends buckets to correct processes.

– Phase 4: each process sorts its received bucket. Final result is globally sorted.

– Used in: MPI parallel sort libraries, Apache Spark sort, distributed sort in databases.

13.5 Parallel Matrix Multiplication

• **Cannon's Algorithm** (2D process torus):

– Partition A and B into √P × √P blocks. Pre-shift: row i of A shifted left i steps; column j of B shifted up j steps.

– In each of √P steps: multiply local blocks (C += A_local × B_local); shift A one step left; shift B one step up.

– After √P steps, each process has accumulated its correct C block.

– Communication: O(N²/√P) per step. Memory: O(N²/P) per process — optimal for matrix multiply.

• **SUMMA Algorithm** (ScaLAPACK basis):

  – Broadcasts one block-column of A and block-row of B at a time; rank-k update C += A_col × B_row.
  – More numerically stable than Cannon's; supports non-square process grids; easier to implement correctly.
  – MPI_Bcast along processor rows and columns handles communication.

• **Parallel FFT**:
  – N-point FFT: W=O(N log N), D=O(log N). Parallelism = N.
  – Communication: all-to-all at each of log N stages — bisection bandwidth critical.
  – Butterfly communication pattern maps naturally to hypercube and fat tree topologies.

14. Parallel I/O

Parallel I/O enables simultaneous access to storage by multiple processes, preventing I/O from becoming the application bottleneck as computational throughput scales with processor count.

14.1 The I/O Bottleneck

• Single HDD: ~200 MB/s sequential, ~5ms random access. Single NVMe SSD: ~7 GB/s sequential.

• A 10,000-node cluster producing 1 MB/s output per node = 10 GB/s aggregate — far exceeds any single storage device. Parallel file systems stripe data across hundreds of devices to deliver aggregate bandwidth.

• I/O strategies compared:

– **N files per N processes**: each process writes its own file. No contention; near-linear scaling. Problem: 100,000 files stress filesystem metadata server; downstream tools expect single file.

– **Single shared file, independent I/O**: all processes write to one file at different offsets. Simple; moderate performance; lock contention at high process counts.

– **Collective I/O (MPI-IO)**: library reorganizes many small non-contiguous requests into a few large contiguous I/O operations. Matches filesystem's preferred access granularity. Best performance.

14.2 MPI-IO — Two-Phase I/O

• Two-phase I/O converts N small non-contiguous writes into a few large contiguous writes:

– **Phase 1 (data exchange)**: each process sends its data to the process that owns the relevant file region, using all-to-all communication.

– **Phase 2 (I/O phase)**: each I/O aggregator process writes one large contiguous block to the file.

– Converts random-access pattern → sequential large writes. Eliminates disk seek time; matches Lustre stripe boundaries.

• Key MPI-IO functions:

– **MPI_File_open(comm, filename, amode, info, &fh;)**: open file collectively.

– **MPI_File_set_view(fh, disp, etype, filetype, ...)**: each process sets its view — which portions of the file it owns, described by MPI derived datatypes.

– **MPI_File_write_all(fh, buf, count, type, &status;)**: collective write — all processes participate simultaneously; library aggregates.

– **MPI_File_read_at(fh, offset, buf, count, type, &status;)**: independent read at explicit byte offset.

14.3 Parallel File Systems

• **Lustre** — dominant HPC parallel file system (70% of TOP500):

– Separates metadata (MDS/MDT: file creation, directory listing) from data (OSS/OST: actual file data).

– File striped across multiple OSTs in 'stripe_count' × 'stripe_size' chunks (default 1 MB per OST).

– 1,000 clients writing a file striped across 50 OSTs → 50× single-OST bandwidth. Petabyte capacities; hundreds of GB/s aggregate bandwidth.

• **HDFS (Hadoop Distributed File System)**:

– Files split into 128 MB blocks; each block replicated 3× across DataNodes for fault tolerance.

– NameNode (single master): stores metadata, block locations. DataNode: stores actual blocks.

– Write-once, append-only — no random writes. Optimized entirely for large sequential reads.

– Data locality scheduling: Hadoop/Spark scheduler runs computation on the node that already holds the data block, eliminating network transfer.

• **Ceph**:

– Unified object, block, and POSIX file storage in one system.

– CRUSH algorithm: deterministically computes object placement from object ID + cluster map. No central lookup table $\rightarrow$ no metadata bottleneck.

– MON (monitors): maintain cluster map consensus. OSD (Object Storage Daemons): store data. MDS: handle POSIX metadata for CephFS.

15. Performance Analysis and Tuning

Performance analysis identifies bottlenecks by measuring where time is spent. Tuning applies targeted optimizations — algorithmic, code-level, and runtime — to eliminate identified bottlenecks.

15.1 Roofline Model

- Visual framework (Williams, Waterman, Patterson 2009) showing which hardware resource limits a kernel's performance.
- X-axis: **Arithmetic Intensity (AI)** = FLOPS executed / bytes transferred from DRAM.
- Y-axis: **Performance** in GFLOPS/s.
- Two ceilings:
  - **Compute ceiling**: horizontal line at peak FLOPS/s of the processor.
  - **Memory bandwidth ceiling**: diagonal line with slope = peak memory bandwidth (GB/s). Performance = BW × AI.
- Achievable performance = min(peak FLOPS, BW × AI).
- **Ridge point** = peak FLOPS / peak BW. Example: 10 TFLOPS / 300 GB/s = 33 FLOPs/byte.
  - AI < ridge point: **memory-bandwidth-bound**. Optimizing FLOPS count doesn't help; reduce memory traffic.
  - AI > ridge point: **compute-bound**. Reducing FLOPS or improving vectorization helps; memory optimization doesn't.
- Kernel AI examples:
  - Dense DGEMM (large N): AI ≈ N/3 FLOPs/byte — heavily compute-bound.
  - SpMV (sparse matrix-vector): AI ≈ 0.5–2 FLOPs/byte — severely memory-bandwidth-bound.
  - Jacobi 2D stencil: AI ≈ 1.7 FLOPs/byte — memory-bound; loop blocking dramatically improves AI.

15.2 Profiling Tools

Tool	Type	Overhead	Key Capability
gprof	Sampling + instrumentation	1–5%	Flat profile + call graph; compile with -pg
perf stat	Hardware counter sampling	< 1%	Cache miss rates, IPC, branch mispredictions
Callgrind (Valgrind)	Cache simulation	20–100×	Exact cache miss counts per function; no recompile
TAU	Parallel trace + profiling	5–15%	MPI + OpenMP + CUDA; timeline and call graph
Score-P + Vampir	Trace generation + visualization	5–20%	Per-thread timeline: identifies load imbalance, barriers
Intel VTune	Sampling + hardware counters	1–5%	Hotspots, memory access, threading, GPU analysis
NVIDIA Nsight Systems	GPU+CPU timeline	2–5%	CUDA kernel, memory transfer, CPU-GPU overlap
NVIDIA Nsight Compute	GPU kernel deep-dive	5–50×	Warp efficiency, memory throughput, occupancy, roofline

15.3 Common Bottlenecks and Fixes

- **Memory bandwidth saturation**:
  - Symptom: achieved bandwidth close to peak; performance doesn't improve with more cores.

– Fix: loop tiling/blocking to improve cache reuse. Switch Array-of-Structures (AoS) → Structure-of-Arrays (SoA) for coalesced access. Reduce data precision (FP64 → FP32 or FP16) to double/quadruple bandwidth.

• **Communication overhead**:

– Symptom: trace shows long MPI_Wait or MPI_Barrier times while other processes are computing.

– Fix: use non-blocking MPI (Isend/Irecv) and overlap with computation. Reduce message count via collective operations. Reduce volume by compressing halo data or sending only changed values.

• **Load imbalance**:

– Symptom: trace shows some processes active much longer than others; others wait in barriers.

– Fix: switch from static block partitioning to dynamic scheduling. Use graph partitioning (METIS) for irregular mesh. Apply work stealing at runtime.

• **False sharing**:

– Symptom: parallel efficiency collapses at 2–4 threads; perf shows high cache miss rates on coherence traffic counters.

– Fix: pad thread-private data structures to 64-byte (one cache line) boundaries using alignas(64).

• **Excessive synchronization**:

– Symptom: many fine-grained locks or barriers; parallel section faster than expected but synchronization dominates.

– Fix: reduce barrier frequency (use nowait), coarsen critical sections, use atomic operations instead of mutexes, redesign to eliminate sharing.

16. Power in Parallel Systems

Power consumption is a first-order constraint in parallel system design. A petascale HPC cluster consumes 20–40 MW — comparable to a small city. Power costs often exceed hardware costs over a system's lifetime.

16.1 Power Components

- **Dynamic power**: consumed by transistors switching state. Formula:
  - **$P_{dynamic} = \alpha \times C \times V^2 \times f$**
  - α = activity factor (fraction of transistors switching per cycle, typically 0.1–0.5).
  - C = total switching capacitance. V = supply voltage. f = clock frequency.
  - Key insight: $P \propto V^2$. Reducing voltage by 10% reduces dynamic power by 19%.
  - $P \propto f$, and since $V \propto f$ (minimum voltage grows with frequency): $P_{total} \propto f^3$. Cubic relationship. Halving frequency reduces power by 8×.
- **Static power (leakage)**: consumed even when transistors are not switching.
  - Caused by sub-threshold leakage current through gate oxide. Increases as transistors shrink below 10 nm.
  - Modern solution: **power gating** — cut power supply to idle cores entirely. Used in mobile and server CPUs.
- **Cooling power**: significant fraction of total facility power.
  - Air cooling: CRAC units consume 30–50% of total facility power just for cooling.
  - Liquid cooling: 1,000× more thermally efficient than air. Reduces cooling to 5–10% of IT power.
  - Immersion cooling: entire servers submerged in dielectric fluid. Highest efficiency; used in HPC and cryptocurrency mining.

16.2 DVFS — Dynamic Voltage and Frequency Scaling

- DVFS reduces power during low-demand periods by simultaneously reducing both voltage and clock frequency.
- Operation sequence matters:
  - Increasing frequency: raise voltage FIRST (ensure adequate voltage before speeding up).
  - Decreasing voltage: reduce frequency FIRST (slow down before reducing voltage to avoid timing violations).
- Linux cpufreq governors:
  - **performance**: always maximum frequency. Used for HPC workloads.
  - **powersave**: always minimum frequency.
  - **ondemand**: monitors CPU utilization; adjusts every 10–100 ms.
  - **schedutil**: integrates with Linux CFS scheduler; per-task frequency decisions — faster response than ondemand.
- HPC consideration: reducing frequency saves dynamic power but extends runtime, increasing static power consumed. For compute-bound workloads at maximum frequency, total energy may actually be LOWER than at reduced frequency because shorter runtime reduces static leakage exposure.
- **PUE** (Power Usage Effectiveness) = Total facility power / IT equipment power.
  - Ideal PUE = 1.0 (no overhead). Typical data center: 1.2–1.5. Older facilities: 2.0+.

– Google data centers: average PUE ~1.1. Meta: ~1.08. Top HPC centers target < 1.15.

• **Green500 List**: ranks TOP500 supercomputers by energy efficiency in GFLOPS/Watt. Frontier (1.1 exaFLOP): ~52 GFLOPS/Watt.

17. Programming Models

A parallel programming model is an abstraction specifying how parallelism is expressed, how tasks communicate and synchronize, and how tasks map to hardware. The model chosen dictates program structure, correctness complexity, performance, and portability.

17.1 Data Parallel Model

• Same operation applied to every element of a large dataset simultaneously. Programmer specifies the operation; runtime maps to processors.

• Variants:

– **SPMD**: each MPI process runs same program on different data partition. Most common in HPC.

– **Loop parallelism** (OpenMP): loop iterations distributed across threads; each runs same loop body on different indices.

– **Array programming** (NumPy, Julia, Fortran 90): A = B + C operates element-wise implicitly in parallel.

– **CUDA data-parallel kernel**: same kernel code runs on thousands of threads each with unique global ID to access its data element.

• Best for: regular, dense computations — matrix operations, stencil codes, image processing, signal processing.

17.2 Task Parallel Model

• Different tasks (distinct functions) execute concurrently, potentially on different data. Tasks may be independent or connected by data dependencies.

• **Task DAG**: directed acyclic graph of task dependencies. Critical path = minimum possible runtime; average parallelism = W/D = max useful processors.

• Implementations:

– **OpenMP tasks**: #pragma omp task spawns asynchronous work; #pragma omp taskwait synchronizes.

– **Cilk**: cilk_spawn / cilk_sync. Proven work-stealing scheduler guarantees E[T] = W/P + O(D).

– **Intel TBB**: task_group, parallel_invoke, pipeline constructs.

– **Java Fork/Join**: ForkJoinPool with RecursiveTask for divide-and-conquer.

• Best for: irregular problems — recursive algorithms, tree traversal, adaptive mesh refinement, graph algorithms.

17.3 MapReduce and Spark

• **MapReduce** (Dean & Ghemawat, Google 2004): programmer provides map() and reduce() functions; framework handles all parallelism, distribution, fault tolerance, I/O.

– **Map(key, value)** → list of (key', value') intermediate pairs. Applied to all input records in parallel.

– **Shuffle and Sort**: group all intermediate pairs by key. Sorts and routes pairs to correct reduce tasks via network. Most expensive phase.

– **Reduce(key', list(value'))** → final output. Aggregates all values for a key.

• **Apache Spark**: addresses Hadoop's disk I/O limitation by keeping intermediate results in memory.

– **RDD** (Resilient Distributed Dataset): immutable, partitioned collection with lineage graph for recomputation on failure.

– For iterative algorithms (ML, graph): Spark 10–100× faster than Hadoop — data stays in DRAM across iterations instead of writing to HDFS after every stage.

– APIs: Spark SQL, Spark Streaming, MLlib, GraphX. Languages: Python (PySpark), Scala, Java, R.

17.4 Hybrid MPI+OpenMP Model

• Combines MPI (between nodes) with OpenMP (within each node). One MPI process per node; multiple OpenMP threads per process.

• Advantages:

– Reduces MPI process count: fewer MPI communicators, smaller collective operation overhead.

– Avoids replicating read-only data: one MPI process per node loads data once; all threads share it.

– Better NUMA utilization: one MPI process per NUMA node, threads pinned to same NUMA domain.

• Disadvantages:

– More complex code: two levels of parallelism to tune and debug simultaneously.

– MPI thread safety required: MPI_Init_thread with MPI_THREAD_FUNNELED (only master thread calls MPI) or MPI_THREAD_MULTIPLE (any thread may call MPI).

18. Scalability and Performance Studies

Scalability analysis characterizes how performance and efficiency change as processors and/or problem size increase. It determines whether investing in more hardware delivers proportional benefit.

18.1 Strong vs Weak Scaling

- **Strong scaling**: fixed problem size; increase P. Goal: minimize time.
  - Efficiency E(P) = T(1) / (P × T(P)) decreases as P grows — Amdahl's Law applies.
  - Measured by: plot T vs P. Ideal: hyperbolic. Reality: flattens due to serial fraction and communication overhead.
- **Weak scaling**: problem size scales proportionally with P; time should remain constant.
  - More meaningful for scientific computing: scientists always want to solve the largest problem that fits in available memory.
  - Measured by: plot T vs P at constant work per processor. Ideal: flat. Reality: rises slowly due to communication and synchronization overhead that grows with P.
- **Isoefficiency analysis**:
  - Function W(P) giving minimum work to maintain constant efficiency E as P grows.
  - Smaller isoefficiency function = more scalable algorithm.
  - W(P) = O(P): excellent scalability (perfect — work grows linearly with processors).
  - W(P) = O(P log P): good scalability. W(P) = O(P^2): poor — requires quadratic problem size growth.

18.2 HPC Benchmarks

- **LINPACK / HPL**: solve dense N×N linear system Ax=b via LU factorization. Measures peak floating-point throughput. Drives TOP500 ranking.
  - Memory access pattern is regular and cache-friendly — near-peak compute utilization achievable.
  - Criticized for not representing real workloads; most applications achieve far less than LINPACK performance.
- **HPCG** (High Performance Conjugate Gradient): sparse PCG solve on 3D Laplacian.
  - Memory-bandwidth-bound, communication-intensive — far better proxy for real scientific codes.
  - Most systems achieve only 1–10% of their LINPACK score on HPCG. Exposes memory and interconnect limitations.
- **Graph500**: measures graph traversal rate in GTEPS (Giga Traversed Edges Per Second). Tests memory latency and random access patterns — fundamentally different bottleneck from LINPACK.
- **IO500**: measures parallel I/O performance (bandwidth and metadata operations per second). Increasingly important as data volumes grow.

19. Scheduling in Parallel & Distributed Systems

Scheduling assigns tasks or jobs to processing resources over time. Good scheduling maximizes resource utilization, minimizes completion time, respects dependencies and resource constraints, and ensures fairness across users.

19.1 Task Graph Scheduling

• **Task DAG**: nodes = tasks with computation time; directed edges = precedence constraints (B depends on A's output).

• **Critical path**: longest weighted path from source to sink. No schedule can complete before critical path length — hard lower bound on makespan regardless of processor count.

• **List scheduling** (2-approximation algorithm):

– Assign priority to each task, e.g., **bottom level** = critical path from this task to exit.

– Maintain sorted ready list of tasks whose predecessors have all completed.

– Assign highest-priority ready task to earliest available processor.

– Achieves at most 2 – 1/P times optimal makespan. Optimal scheduling is NP-hard in general.

• **Heterogeneous Earliest Finish Time (HEFT)**: widely used for heterogeneous systems.

– Compute upward rank for each task: rank_u(n) = computation_cost(n) + max over successors (comm_cost + rank_u(successor)).

– Sort tasks by decreasing rank_u. Assign each task to the processor that gives earliest finish time.

– More sophisticated than basic list scheduling; accounts for heterogeneous processor speeds and network costs.

19.2 HPC Job Scheduling — SLURM

• SLURM (Simple Linux Utility for Resource Management): dominant workload manager for large HPC clusters.

• Users submit batch scripts specifying: nodes, cores/node, GPUs, memory, wall-clock time limit, QoS tier, dependencies.

• SLURM assigns jobs to partitions (queues) with different priorities and resource limits.

• **Backfill scheduling**: allows small lower-priority jobs to start immediately if they complete before a pending high-priority large job's required resources become available.

– Keeps cluster busy (high utilization) while not delaying high-priority jobs.

– Without backfill: large jobs leave idle nodes sitting unused while waiting for remaining nodes to drain.

• **Fair-share scheduling**: recent resource consumption affects job priority.

– Group/user that consumed MORE than fair share recently: lower priority.

– Group/user that consumed LESS: higher priority. Over time, each group receives its allocation.

• SLURM commands:

– **sbatch job.sh**: submit batch job. **squeue**: view job queue. **scancel jobid**: cancel job.

– **sinfo**: view cluster state. **sacct**: view job accounting. **srun**: run interactive job.

19.3 Cloud and Grid Scheduling

• **HTCondor**: high-throughput computing; harvests idle workstation/server cycles across administrative domains.

  – ClassAds: flexible resource requirement matching via arbitrary constraint expressions.

  – DAGMan: workflow manager for submitting jobs with inter-job dependencies.

• **Kubernetes**: container orchestration with pod scheduling based on resource requests/limits.

  – Scheduler considers: node available CPU/memory, affinity rules, taints/tolerations, PriorityClass.

  – Bin-packing: fits as many pods on fewest nodes to minimize cluster footprint.

• **Globus**: grid computing middleware for research institutions.

  – Globus Transfer: high-performance file transfers between institutions with automatic checkpointing/retry.

  – Globus Flows: multi-step automated data movement and processing pipelines across multiple sites.

20. Storage Systems

Distributed storage systems provide scalable, reliable, high-performance data persistence across many nodes, presenting a coherent storage namespace despite data distributed across thousands of physical drives.

20.1 Storage Architecture Types

Type	Full Name	Level	Description	Use Case
DAS	Direct-Attached Storage	Block	Storage directly connected to server; not shared over network	Single-server databases, local NVMe
NAS	Network-Attached Storage	File	File-level storage over TCP/IP (NFS, SMB)	Shared home directories, office files
SAN	Storage Area Network	Block	Block-level storage over FC or iSCSI; high IOPS	Databases requiring raw block devices
Object Storage	–	Object	Data as objects with metadata + URL; HTTP REST API	Cloud storage, big data, media archives
Parallel FS	Lustre, GPFS	File	Stripes data across many OSTs; HPC I/O	HPC simulation output, checkpoint files

20.2 RAID Levels — Complete

RAID	Name	Min Disks	Fault Tolerance	Write Overhead	Storage Efficiency	Best For
0	Striping	2	None (total loss on 1 failure)	None	100%	Max performance, no redundanc
1	Mirroring	2	1 disk failure	2× writes	50%	Critical data, fast recovery
5	Stripe+Parity	3	1 disk failure	Parity recalc on write	(N–1)/N	General purpose NAS/SAN
6	Double Parity	4	2 simultaneous failures	More write overhead	(N–2)/N	Large arrays where 2-disk failure
10	Stripe+Mirror	4	1 per mirror set	2× writes	50%	High-performance databases

20.3 CAP Theorem

• CAP theorem (Brewer 2000; proved Gilbert & Lynch 2002): a distributed storage system can provide at most 2 of 3 properties simultaneously.

• The three properties:

  – **Consistency (C)**: every read returns the most recent write or an error. All nodes see the same data at the same time.

  – **Availability (A)**: every request receives a non-error response. System always responds, even if data may be stale.

  – **Partition Tolerance (P)**: system continues operating despite arbitrary network message loss or delay between nodes.

• Network partitions are unavoidable in real distributed systems. Therefore P is non-negotiable — every real system must be CP or AP.

• **CP systems**: sacrifice availability during partitions to maintain consistency.

  – Examples: Zookeeper, HBase, Etcd, MongoDB (strong consistency mode).

  – Behavior: refuse reads/writes during partition rather than serve potentially inconsistent data.

• **AP systems**: sacrifice consistency to remain available during partitions.

  – Examples: Cassandra, DynamoDB, Riak, CouchDB.

– **Eventual consistency**: all nodes will converge to the same value when the partition heals, but during partition may return stale data.

• **PACELC extension** (Abadi 2012): even without partition (normal operation), must trade off Latency vs Consistency.

– Serve from nearest replica immediately (low latency, possibly stale) OR contact quorum of replicas (higher latency, consistent).

– This L-C trade-off exists always, not just during failures — real systems tune read/write quorums accordingly.

21. Synchronization

Synchronization coordinates the timing and ordering of operations across concurrent processes to maintain correctness invariants, prevent races, and ensure proper visibility of shared state updates.

21.1 Lamport Logical Clocks

• Problem: distributed systems have no shared physical clock. Need a way to order events without synchronizing wall clocks.

• Lamport (1978) defined the **happens-before (→)** relation:

– a → b if a and b are in the same process and a precedes b in program order.

– a → b if a is the sending of a message and b is the receipt of that same message.

– a → b by transitivity if ∃c: a → c and c → b.

– a ■ b (concurrent): neither a → b nor b → a.

• Lamport timestamp assignment:

– Each process maintains counter C, initialized to 0.

– Before each event: increment C.

– Send: include current C as timestamp in message.

– Receive message with timestamp T: set C = max(C, T) + 1.

• Guarantee: if a → b then L(a) < L(b). But NOT conversely — L(a) < L(b) does NOT imply a → b. Lamport clocks cannot distinguish causality from coincidence.

21.2 Vector Clocks

• Vector clocks (Fidge & Mattern, 1988): each process i maintains vector V_i[0..N–1] of N counters, one per process.

• Update rules:

– Before each event at process i: increment V_i[i].

– On send: include entire vector V_i.

– On receive at process j with vector V_msg: for each k, V_j[k] = max(V_j[k], V_msg[k]); then increment V_j[j].

• Comparison:

– V(a) ≤ V(b) (component-wise) with at least one strict < : a causally precedes b.

– Neither V(a) ≤ V(b) nor V(b) ≤ V(a): a and b are concurrent (causally independent).

• Applications:

– Distributed debugging: precisely identify which events are causally related.

– Version control: Git uses vector-clock-like logic to detect merge conflicts.

– Conflict detection: Dynamo/Riak use vector clocks to detect concurrent writes needing merge resolution.

21.3 Chandy-Lamport Distributed Snapshot Algorithm

• Goal: record a consistent global state (snapshot) of a running distributed computation WITHOUT stopping it.

• Consistent snapshot: no message appears as received without also appearing as sent — the snapshot must represent a state that could actually have occurred.

• Algorithm steps when process P■ initiates snapshot:

  – P■ records its own local state.

  – P■ sends a special MARKER message on ALL outgoing channels before any other messages.

• When process P■ receives a MARKER on channel C_ij for the **first time**:

  – P■ records its own local state.

  – Records C_ij's state as EMPTY (no messages were in transit when marker arrived).

  – P■ sends MARKERs on ALL its outgoing channels.

• When P■ receives a MARKER on channel C_kj **after already recording its state**:

  – Records C_kj's state as: all messages received on C_kj since recording local state (these were in-transit when snapshot was taken).

• Termination: all processes have recorded their state and all channels have been recorded.

• Applications: distributed deadlock detection, distributed debugging, checkpointing for fault recovery, global property verification.

21.4 Lock-Free and Wait-Free Synchronization

• Lock-free: at least one thread makes progress even if others are arbitrarily delayed or fail.

  – Contrast with mutexes: if a thread holding a mutex crashes, all threads waiting on it are permanently blocked.

  – Lock-free structures survive thread delays without global blocking.

• Wait-free: every thread completes its operation in a bounded number of steps, regardless of other threads' behavior. Stronger than lock-free.

• Built on atomic primitives:

  – **CAS (Compare-and-Swap)**: if memory[addr] == expected, write new value; return old value. Universally available (x86: CMPXCHG; ARM: LDXR/STXR loop; Java: AtomicInteger.compareAndSet).

  – **Fetch-and-Add**: atomically add and return old value. Used in lock-free queues and ticket locks.

  – **LL/SC (Load-Linked / Store-Conditional)**: ARM/RISC-V native; SC fails if any other write to the address occurred since the LL.

• Lock-free data structures:

  – **Lock-free stack**: CAS on head pointer for push/pop. Risk: ABA problem (address reused — use stamped pointers or hazard pointers).

  – **Michael-Scott lock-free queue**: separate head/tail; CAS-based enqueue/dequeue. Used in Java's ConcurrentLinkedQueue.

22. Tools Reference

The PDC ecosystem includes programming libraries, runtime environments, cluster middleware, debugging tools, and cloud platforms used for developing, deploying, and optimizing parallel and distributed applications.

22.1 CUDA — Full Ecosystem

- NVIDIA's proprietary GPU parallel computing platform. Extends C/C++/Fortran with GPU execution model.
- **nvcc compiler**: separates GPU (device) code from CPU (host) code. Compiles device code to PTX (virtual ISA), then JIT-compiled by driver to native machine code for target GPU architecture.
- Key CUDA libraries (all NVIDIA-optimized, near-peak performance):
  - **cuBLAS**: all BLAS levels. DGEMM achieves >90% peak FLOPs on H100 using Tensor Cores.
  - **cuFFT**: multi-dimensional FFTs with automatic plan selection for array dimensions.
  - **cuDNN**: deep learning primitives — convolution, pooling, batch normalization, attention.
  - **cuSPARSE**: sparse matrix operations (SpMM, SpMV, conversion between sparse formats).
  - **Thrust**: STL-like parallel algorithms (sort, reduce, scan, transform) for both GPU and CPU.
- Profiling and debugging:
  - **Nsight Systems**: CPU+GPU timeline — kernel execution, memory transfers, NVTX markers, CPU-GPU overlap visualization.
  - **Nsight Compute**: kernel-level deep dive — warp efficiency, memory throughput, roofline, source-to-PTX-to-SASS analysis.
  - **cuda-gdb**: debug CUDA kernels with breakpoints, watchpoints, thread/warp inspection.
  - **compute-sanitizer**: detects memory errors (out-of-bounds, races) in GPU kernels.

22.2 OpenMP — Advanced Features Summary

Directive / Clause	Effect	Key Notes
#pragma omp parallel	Create thread team	Default: shared variables; declare privates explicitly
#pragma omp for	Distribute loop iterations	schedule(static/dynamic/guided/runtime) clause
nowait clause	Remove implicit barrier at end of for	Only safe if next phase doesn't depend on prior writes
#pragma omp sections	Assign code blocks to different threads	Each section runs on one thread
#pragma omp critical	Mutual exclusion for code block	Named critical: #pragma omp critical(name)
#pragma omp atomic	Atomic single-variable update	Faster than critical; limited to read/write/update/capture
reduction(op:var)	Private copies combined at end	Eliminates race on accumulation variable
#pragma omp task	Asynchronous task creation	For irregular parallelism; depends() for task DAG
#pragma omp target	Offload to accelerator (GPU)	Requires map() clauses for data movement
OMP_PROC_BIND	Thread-to-core binding policy	close/spread/master; critical for NUMA performance

22.3 MPICH and OpenMPI

- **MPICH**: ANL's reference MPI implementation. Basis for most vendor-optimized MPI libraries.
  - Intel MPI: optimized for Intel Xeon + Omni-Path/InfiniBand.

– Cray MPICH: tuned for Cray Slingshot interconnect; used on Frontier and Aurora.

– IBM Spectrum MPI: optimized for IBM POWER + NVIDIA GPU (NVSHMEM integration).

• **OpenMPI**: open-source combining FT-MPI, LA-MPI, LAM/MPI. UCX (Unified Communication X) transport layer handles InfiniBand, RoCE, TCP, shared memory automatically.

– Modular architecture: OFI (Open Fabrics Interface), UCX, TCP components selectable at runtime.

– Process binding: --map-by numa --bind-to core pins each MPI process to a NUMA node core for optimal memory locality.

• MPI thread safety levels (required for hybrid MPI+OpenMP):

– **MPI_THREAD_SINGLE**: only one thread total.

– **MPI_THREAD_FUNNELED**: multiple threads but only master calls MPI.

– **MPI_THREAD_SERIALIZED**: any thread may call MPI but not simultaneously.

– **MPI_THREAD_MULTIPLE**: any thread may call MPI at any time. Required for fully threaded MPI.

22.4 Hadoop Ecosystem

• **YARN** (Yet Another Resource Negotiator): Hadoop 2+ cluster resource manager.

– ResourceManager: tracks all cluster resources (CPU, memory per node).

– ApplicationMaster: per-job manager; negotiates resources; schedules tasks on NodeManagers.

– NodeManager: per-node agent; launches/monitors containers; reports resources to ResourceManager.

– Multiple frameworks (MapReduce, Spark, Flink) share the same YARN cluster simultaneously.

• **Hive**: SQL on Hadoop. HiveQL → MapReduce or Tez execution plans.

– ORC/Parquet columnar formats: predicate pushdown skips entire column chunks based on min/max statistics.

– Vectorized reading: processes thousands of rows per batch instead of one at a time.

• **HBase**: distributed column-oriented NoSQL database (Google Bigtable-inspired).

– Random real-time read/write access to billion-row tables.

– Architecture: RegionServers hold data partitions (Regions); HMaster coordinates. ZooKeeper for distributed coordination.

– Write path: WAL (Write-Ahead Log) → MemStore (in-memory) → periodic flush to HFile on HDFS.

• **Apache Spark**: in-memory distributed computation.

– **RDD**: immutable partitioned collection with lineage for recomputation. Created by SparkContext.parallelize() or reading from HDFS.

– Transformations (map, filter, join): lazy — build execution plan. Actions (count, collect, save): trigger execution.

– DAG scheduler: optimizes execution plan; pipelines transformations; avoids unnecessary shuffles.

22.5 Cilk, HTCondor, Globus

• **Cilk / OpenCilk**:

– **cilk_spawn**: launch child function as potentially parallel task. Parent continues; child runs when scheduler decides.

– **cilk_sync**: wait for all cilk_spawn'd children in current function scope to complete.

– Work-stealing scheduler: expected runtime = W/P + O(D). Space bound: O(S × P) where S = serial space.

• **HTCondor**:

– Harvests idle CPU cycles on desktop workstations and servers across an institution.

– ClassAds: job advertises requirements (OS, memory, software); machine advertises capabilities; flexible expression-based matching.

– DAGMan: submit workflows with inter-job dependencies as a DAG; automatically handles retries on failure.

– Checkpointing support: can checkpoint and migrate jobs between machines.

• **Globus**:

– Globus Transfer: fire-and-forget file transfers between research sites. Optimizes TCP parameters, handles checkpointing/restart automatically.

– GridFTP underlying protocol: parallel multi-stream TCP with extended mode for high-throughput WAN transfers.

– Globus Auth: federated identity and authorization across institutions using OAuth2/OpenID Connect.

22.6 gdb, FUSE, threads, Swift, AWS, OpenStack

Tool	Category	Purpose and Key Features
gdb	Debugger	GNU debugger. MPI: attach to individual processes. Thread cmds: info threads, threa
FUSE	FS Framework	Filesystem in Userspace. Implement custom FS without kernel module. Uses: SSHFS
POSIX threads	Library	pthread_create, pthread_join, pthread_mutex_lock/unlock, pthread_cond_wait/signal.
C++11 threads	Library	std::thread, std::mutex, std::condition_variable, std::atomic<T>, std::future/promise. P
Swift (parallel)	Scripting	Swift/T: parallel scripting for multi-site workflows. NOT Apple's Swift. Compiles Swift s
Amazon AWS	Cloud Platform	EC2 Hpc instances, EFA (200 Gb/s low-latency network for MPI), AWS ParallelCluste
OpenStack	Open-source Cloud	IaaS cloud OS. Nova (VM compute), Swift (object storage), Neutron (networking), Cin

Quick Reference — Formulas, Laws, Complexity

Core Performance Laws

Formula	Name	Notes
S(P) = T(1) / T(P)	Speedup	P = processors
E(P) = S(P) / P × 100%	Parallel Efficiency	100% = perfect; 60–80% = typical good
S(P) = 1/(f + (1–f)/P)	Amdahl's Law	f = serial fraction; max S = 1/f
S_scaled = P – f(P–1)	Gustafson's Law	f = serial fraction at scale
T(P) ≥ max(W/P, D)	Work-Depth bound	W = total work; D = critical path
Perf ≤ min(peak FLOPS, BW × AI)	Roofline	AI = FLOPs / DRAM bytes
AMAT = t_hit + MR × t_miss	Avg Memory Access Time	MR = cache miss rate
t_e = (O + 4M + P) / 6	PERT expected duration	O=optimistic, M=most likely, P=pessimistic
A = MTTF / (MTTF + MTTR)	System Availability	5 nines = 99.999% = 5.26 min/year downtime

CUDA Thread Indexing

- 1D: global_id = blockIdx.x × blockDim.x + threadIdx.x
- 2D: row = blockIdx.y × blockDim.y + threadIdx.y | col = blockIdx.x × blockDim.x + threadIdx.x
- Warp size = 32 threads (fixed, all NVIDIA GPUs)
- Occupancy = active warps per SM / maximum warps per SM

MPI Collective Cost (Hockney: T = α + β×n)

Collective	Time Complexity	Note
MPI_Bcast	O(log P × (α + β×n))	Tree broadcast
MPI_Reduce	O(log P × (α + β×n))	Tree reduce to root
MPI_Allreduce	O(2 log P × (α + β×n))	Reduce + broadcast
MPI_Scatter / Gather	O(log P × α + β×n)	–
MPI_Alltoall	O(P×α + β×n)	Most expensive; bottleneck in FFT/transpose
MPI_Barrier	O(2 log P × α)	No data transferred

RAID Quick Reference

Level	Name	Fault Tolerance	Storage Efficiency
0	Striping	None	100%
1	Mirroring	1 disk	50%
5	Stripe+Parity	1 disk	(N–1)/N
6	Double Parity	2 disks	(N–2)/N

10	Stripe+Mirror	1 per mirror set	50%

Topology Quick Reference

Topology	Diameter	Bisection Width	Degree
Ring	N/2	2	2
2D Mesh	2(√N–1)	√N	≤4
2D Torus	√N	2√N	4
Hypercube	log■N	N/2	log■N
Fat Tree	2 log(N/k)	N/2	k (switch-dependent)
Dragonfly	3	~N/2	High, fixed

End of Notes — Parallel and Distributed Computing